2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Comparison of Random Number Generators in Particle Swarm Optimization Algorithm

Ke Ding and Ying Tan

Abstract—**Intelligent optimization algorithms are very effective to tackle complex problems that would be difficult or impossible to solve exactly. A key component within these algorithms is the random number generators (RNGs) which provide random numbers to drive the stochastic search process. Much effort is devoted to develop efficient RNGs with good statistical properties, and many highly optimized libraries are ready to use for generating random numbers fast on both CPUs and other hardware platforms such as GPUs. However, few study is focused on how different RNGs can effect the performance of specific intelligent optimization algorithms. In this paper, we empirically compared 13 widely used RNGs with uniform distribution based on both CPUs and GPUs, with respect to algorithm efficiency as well as their impact on Particle Swarm Optimization (PSO). Two strategies were adopted to conduct comparison among multiple RNGs for multiple objectives. The experiments were conducted on well-known benchmark functions of diverse landscapes, and were run on the GPU for the purpose of accelerating. The results show that RNGs have very different efficiencies in terms of speed, and GPU-based RNGs can be much faster than their CPU-based counterparts if properly utilized. However, no statistically significant disparity in solution quality was observed. Thus it is reasonable to use more efficient RNGs such as Mersenne Twister. The framework proposed in this work can be easily extended to compare the impact of non-uniformly distributed RNGs on more other intelligent optimization algorithms.**

I. INTRODUCTION

Random numbers are widely used in intelligent optimization algorithms such as Genetic Algorithm (GA), Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO), just to name a few. Random numbers are usually generated by deterministic algorithms called Random Number Generators (RNGs) and play a key role in driving the search process.

The performance of RNGs can be analyzed theoretically using criteria such as period and lattice structure [1], [2], or by systematic statistical test [3]. However, none of these analyses are relevant directly to RNGs' impact on optimization algorithms like PSO.

It is interesting to ask how RNGs can effect these stochastic methods. Clerc [4] replaced the conventional RNGs with a short length list of numbers (i.e. a RNG with a very short period) and empirically studied the performance of PSO. The experiments show that, at least for the moment, there is no sure way to build a "good" list for high performance. Thus, RNGs with certain degree of randomness are necessary for the success of stochastic search.

Bastos-Filho et al. [5], [6] studied the impact of the quality of CPU- and GPU-based RNGs on the performance of PSO. The experiments show that PSO needs RNGs with minimum quality and no significative improvements were achieved when comparing high quality RNGs to medium quality RNGs. Only Linear Congruential Generator (LCG) [1] and Xorshift algorithms [7] were compared for CPUs, and only one method for generating random numbers in an ad hoc manner on GPUs was adopted for comparing GPUs.

In general, RNGs shipped with math libraries of programming languages or other specific random libraries are used when implementing intelligent optimization algorithms. These RNGs generate random numbers of very diverse qualities with different efficiency. A comparative study on the impact of these popular RNGs will be helpful when implementing intelligent algorithms for solving optimization problems.

In this paper, we selected 13 widely used, highly optimized uniformly distributed RNGs and applied them to PSO for empirically comparing their impact on the optimization performance. Nine well-known benchmark functions were implemented for the sake of comparison. All the experiments were conducted on the GPU for fast execution. Two novel strategies, league scoring strategy and lose-rank strategy, were introduced to conduct a systematic comparison on these RNGs' performance. Though the work is limited to the impact on PSO, other intelligent algorithms can also be studied in the proposed framework.

The remainder of this paper is organized as follows. The next section presents a brief introduction to RNGs. Special attention is drawn to well-studied and popular algorithms for uniformly distributed random number generation. The experimental setup is described in detail in Section III. Section IV presents the experimental results on both RNGs's efficiency and the performance of PSO with these RNGs. Analyses of the results are also given in this section. We conclude this paper in section V.

II. RANDOM NUMBER GENERATORS

According to the source of randomness, random number generators fall into three categories [8]: true random number generators (TRNGs), quasirandom number generators (QRNGs) and pseudorandom number generators (PRNGs).

TRNGs utilize physical sources of randomness to provide truly unpredictable numbers. These generators are usually slow and unrepeatable, and usually need the support of specialist

Ke Ding and Ying Tan (corresponding author) are with the Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University and Key Laboratory of Machine Perception (Ministry of Education), Peking University, Beijing, 100871, P.R. China. (Email: {keding,ytan}@pku.edu.cn).

This work was supported by the National Natural Science Foundation of China under grants number 61375119, 61170057 and 60875080.

hardware [2]. So TRNGs hardly used in the field of stochastic optimization. QRNGs are designed to evenly fill an n-dimensional space with points. Though quite useful, they are not widely used in the domain of optimization. PRNGs are used to generate pseudorandom sequences of numbers that satisfy most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm. PRNGs are the most common RNGs of the three groups, and provided by almost all programming languages. There also exit many well optimized PRNGs for open access. As we discuss PRNGs in this work, we will use random numbers and pseudorandom numbers alternatively henceforth.

Random numbers can subject to various distributions, such as uniform, normal and cauchy distributions. Of all the distributions, uniform distribution is the most important one. Not only uniform random numbers are widely used in many different domains, but they are used as the base generators for generating random numbers subject to other distributions. Many methods, like transformation methods and rejection methods, can be used to convert uniformly distributed numbers to ones with specific non-uniform distributions [2], [9].

As this work studies uniform distribution only, the remainder of the section merely introduces RNGs with uniform distribution. RNGs for generating uniform distribution random numbers can be classified into two group, according to the basic arithmetic operations utilized: RNGs based on modulo arithmetic and RNGs based on binary arithmetic.

A. *Modulo Arithmetic Based RNGs*

RNGs of this type yield sequences of random numbers by means of linear recurrence modulo m, where m is a large integer.

1) Linear Congruential Generator (LCG): LCG is one of the best-known random number generators. LCG is defined by the following recurrence relation:

$$x_i = a \cdot x_{i-1} + c \mod m$$

where x is the sequence of the generated random numbers and $m > 0$, $0 < a < m$, and $0 \leq c, x_0 < m$. If uniform distribution on $[0, 1)$ is need, then use $u = \frac{x}{m}$ as the output sequence.

For LCG, a, c and m should be carefully chosen to make sure that maximum period can be archived [1]. LCG can be easily implemented on computer hardware which can provide modulo arithmetic by storage-bit truncation. RNG using LCG is shipped with C library (rand()) as well as many other languages such as Java (java.lang.Random). LCG has a relatively short period (at most 2^{32} for 32-bit integer) compared to other more complicated ones.

A special condition of LCG is when $c = 0$, which presents a class of multiplicative congruential generators (MCG) [10]. Multiple carefully selected MCGs can be combined into more complicated algorithms such as Whichmann-Hill generator [11].

2) Multiple Recursive Generator (MRG): MRG is a derivative of LCG and can achieve much longer period. A MRG of order k is defined as follows:

$$x_i = (a_1 \cdot x_{i-1} + a_2 \cdot x_{i-2} + \cdots + a_k \cdot x_{i-k}) \mod m$$

The recurrence has maximal period length $m^k - 1$, if tuple $(a_1, ..., a_k)$ has certain properties [1].

3) Combined Multiple Recursive Generator (CMR): CMR combines multiple MRGs and can obtain better statistical properties and longer periods compared with a single MRG. A well-known implementation of CMR, CMR32k3a [12], combines two MRGs:

$$\begin{aligned} x_i &= a_{11} \cdot x_{i-1} + a_{12} \cdot x_{i-2} + x_{13} \cdot x_{i-3} \mod m_1 \\ y_i &= a_{21} \cdot y_{i-1} + a_{22} \cdot y_{i-2} + x_{23} \cdot y_{i-3} \mod m_2 \\ z_i &= x_i - y_i \mod m_1 \end{aligned}$$

where z forms the required sequence.

B. *Binary Arithmetic Based RNGs*

RNGs of this type are defined directly in terms of bit strings and sequences. As computers are fast for binary arithmetic operations, binary arithmetic based RNGs can be more efficient than modulo arithmetic based ones.

1) Xorshift: Xorshift [7] produces random numbers by means of repeated use of bit-wise exclusive-or (xor, $\oplus$) and shift ($\ll$ for left and $\gg$ for right) operations.

A xorshift with four seeds (x, y, z, w) can be implemented as follows:

$$\begin{aligned} t &= (x \oplus (x_i \ll a)) \\ x &= y \\ y &= z \\ z &= w \\ w &= (w \oplus (w \gg b)) \oplus (t \oplus (t \gg c)) \end{aligned}$$

where w forms the required sequence.

With a carefully selected tuple (a, b, c), the generated sequence can have a period as long as $2^{128} - 1$.

2) Mersenne Twister (MT): MT [13] is one of the most widely respected RNGs, it is a twisted Generalized Feedback Shift Register (GFSR). The underlying algorithm of MT is as follows:

- Set r w-bit numbers (x_i, $i = 1, 2, \cdots, r$) randomly as initial values.
- Let
$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_w & a_{w-1} \cdots a_1 \end{pmatrix},$$
where I_{w-1} is the $(w-1) \times (w-1)$ identity matrix and $a_i, i = 1, \ldots, w$ take values of either 0 or 1. Define
$$x_{i+r} = \left(x_{i+s} \oplus \left(x_i^{(w:(l+1))} | x_{i+1}^{(l:1)}\right) A\right),$$
where $x_i^{(w:(l+1))} | x_{i+1}^{(l:1)}$ indicates the concatenation of the most significant (upper) $w - l$ bits of x_i and the least significant l bits of x_{i+1}.
- Perform the following operations sequentially:

$$\begin{aligned} z &= x_{i+r} \oplus (x_{i+r} \gg t_1) \\ z &= z \oplus ((z \ll t_2) \ \& \ m_1) \\ z &= z \oplus ((z \ll t_3) \ \& \ m_2) \\ z &= z \oplus (x \gg t_4) \\ u_{i+r} &= z/(2^w - 1) \end{aligned}$$

where t_1, t_2, t_3 and t_4 are integers and m_1 and m_2 are bit-masks and '&' is a bit-wise and operation.

$u_{i+r}, i = 1, 2, \cdots$ form the required sequence on interval $(0, 1]$.

With proper parameter values, MT can generate sequence with a period as long as $2^{19,937}$ and extremely good statistical properties [13]. Strategies for selecting good initial values are studied in [14] while Saito et al. [15] proposed efficient implementation for fast execution on GPUs.

III. Experimental Setup

In this section, we describe out experimental environment and parameter settings in detail.

A. Testbed

We conducted our experiments on a PC running 64-bit Windows 7 Professional with 8G DDR3 Memory and Intel core I5-2310 (@2.9 GHz 3.1 GHz). The GPU used for implementing PSO in the experiments is NVIDIA GeForce GTX 560 Ti with 384 CUDA cores. The program was implemented with C and compiled with visual studio 2010 and CUDA 5.5.

B. Particle Swarm Optimization

A standard PSO algorithms [18] with ring topology was adopted in our experiments. Velocity vectors and position vectors are updated with Eq. 1 and Eq. 2, where in Eq. 1, $\omega = 1/(2\log(2)) \approx 0.721$, $c_1 = c_2 = 0.5 + log(2) \approx 1.193$, and r_1, r_2 are random numbers derived from uniform distribution on $(0, 1)$ The swarm size was fixed to 50 for all experiments, and 10,000 iterations was performed for each optimization run.

$$v_{id} = \omega \cdot v_{id} + c_1 r_1 (p_{id} - x_{id}) + c_2 r_2 (p_{gd} - x_{id}) \tag{1}$$

$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{v}_i \tag{2}$$

The PSO algorithm was implemented on the GPU with CUDA based on the work by Zhou et al. [19]. The random numbers generation process was replaced by RNGs under test.

C. RNGs Used for Comparison

Besides functions provided by programming languages, many libraries with well-implemented RNGs are available, such as AMD's ACML [20] and Boost Random Number Library [21] targeted at CPUs and specific implementations ([8], [14], [22]) for GPU platform.

Among all these candidates, Math Kernel Library (MKL) [23] (for CPU) and CURAND [24] (for GPU) were selected for the experiments considering the following reasons: 1) RNGs provided by the two libraries cover the most popular RNG algorithms, and 2) both MKL and CURAND are well-optimized for our hardware platform (I5 CPU and GeForce 560 Ti GPU), so a fair comparison of efficiency can be expected. So experiments with these two libraries are broadly covered in terms of types of RNGs and present a fair comparision in terms of time efficiency.

As LCG is widely shipped by standard library of various programming language, we added a RNG with LCG (C's rand()). The RNGs used in the experiments are list by Tab. I.

D. Benchmark Functions

Nine benchmark functions were implemented on the GPU with float numbers of single precision. All these functions are minimizing problems while $f_1 \sim f_3$ are unimodal function while the left are multimodal functions.

The search space are all limited within $[-10.0, 10.0]^D$, where D is the dimension which could be $10, 30, 50, 100$ in the experiments. The optimum points were shifted to 1.0^D if some where else, and bias values were added to each function to make sure the minimal values are 100 for functions, with the only except of Weierstrass function. Weierstrass function was implemented just as Eq. 8 and no effort was made to move the optima point or adjust the minimal value. The formulas of the used benchmark functions are listed as follows:

Sphere Function

$$f_1 = \sum_{i=1}^{D} \mathbf{x}_i^2 \tag{3}$$

High Conditioned Elliptic Function

$$f_2 = \sum_{i=1}^{D} (10^6)^{\frac{i-1}{D-1}} \mathbf{x}_i^2 \tag{4}$$

Discus Function

$$f_3 = 10^6 \cdot \mathbf{x}_1^2 + \sum_{i=2}^{D} \mathbf{x}_i^2 \tag{5}$$

Rosenbrock Function

$$f_4 = \sum_{i=1}^{D-1} (100 \cdot \left(\mathbf{x}_{i+1} - \mathbf{x}_i^2\right)^2 + (1 - \mathbf{x}_i)^2) \tag{6}$$

Ackley Function

$$f_5 = -20 \cdot \exp(-0.2 \cdot \sqrt{\frac{1}{D} \sum_{i=1}^{D} \mathbf{x}_i^2}) - \exp(\frac{1}{D} \sum_{i=1}^{D} \cos(2\pi \cdot \mathbf{x}_i)) \tag{7}$$

Weierstrass Function

$$f_6 = \sum_{i=1}^{D} (\sum_{k=0}^{20} [0.5^k \cos(2\pi \cdot 3^k (x_i + 0.5))]) \tag{8}$$

Schaffer's F7 Function

$$f_7 = (\frac{1}{D-1} \sum_{i=1}^{D-1} (\sqrt{\mathbf{y}_i} + \sqrt{\mathbf{y}_i} \sin^2(50 \cdot \mathbf{y}_i^{0.2})))^2 \tag{9}$$

where $\mathbf{y}_i = \sqrt{\mathbf{x}_i^2 + \mathbf{x}_{i+1}^2}$

TABLE I: Random Number Generators Tested

No.	Algorithm	Description	Note
1	xorshift	Implemented using the xorshift algorithm [7], created with generator type CURAND_RNG_PSEUDO_XORWOW	CURAND with CUDA Toolkit 5.5
2	xorshift	Same algorithm as 1, faster but probably statistically weaker, set ordering to CURAND_ORDERING_PSEUDO_SEEDED	
3	Combined Multiple Recursive	Implemented using the Combined Multiple Recursive algorithm [12], created with generator type CURAND_RNG_PSEUDO_MRG32K3A	
4	Mersenne Twister	Implemented using the Mersenne Twister algorithm with parameters customized for operation on the GPU [15], created with generator type CURAND_RNG_PSEUDO_MTGP32	
5	Multiplicative Congruential	Implemented using the 31-bit Multiplicative Congruential algorithm [10], create with parameter VSL_BRNG_MCG31	MKL 11.1
6	Generalized Feedback Shift Register	Implemented using the 32-bit generalized feedback shift register algorithms, create with parameter VSL_BRNG_R250 [16]	
7	Combined Multiple Recursive	Implemented using Combined Multiple Recursive algorithm [12], create with parameter VSL_BRNG_MRG32K3A	
8	Multiplicative Congruential	Implemented using the 59-bit Multiplicative Congruential algorithm from NAG Numerical Libraries [11], create with parameter VSL_BRNG_MCG59	
9	Wichmann-Hill	Implemented using the Wichmann-Hill algorithm from NAG Numerical Libraries [11], create with parameter VSL_BRNG_WH	
10	Mersenne Twister	Implemented using the Mersenne Twister algorithm MT19937 [13], create with parameter VSL_BRNG_MT19937	
11	Mersenne Twister	Implemented using the Mersenne Twister algorithms MT2203 [17] with a set of 6024 configurations. Parameters of the generators provide mutual independence of the corresponding sequences., create with parameter VSL_BRNG_MT2203	
12	Mersenne Twister	Implemented using the SIMD-oriented Fast Mersenne Twister algorithm SFMT19937 [14], create with parameter VSL_BRNG_SFMT19937	
13	Linear Congruential	Implementing using Linear Congruential algorithm with $a = 1103515245$, $c = 12345$, $m = 2^{32}$, only high 16 bits are used as output	MS Visual Studio C library rand()

Griewank Function

$$f_8 = \sum_{i=1}^{D} \frac{\mathbf{x}_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{\mathbf{x}_i}{\sqrt{i}}) + 1 \tag{10}$$

Rastrigin Function

$$f_9 = \sum_{i=1}^{D} (\mathbf{x}_i^2 - 10\cos(2\pi \cdot \mathbf{x}_i) + 10) \tag{11}$$

IV. Results and Analysis

This section presents the experimental results. Both efficiency of RNGs and solution quality of PSO using each RNG are described and analyzed.

A. RNGs Efficiency

We ran each RNG program to generate random numbers in batch of different size, and test the speed. The results are presented in Tab. II.

In general, RNGs based on both CPUs and GPUs achieve better performance by generating batches of random numbers, and GPUs need larger batch size to get peek performance than CPUs. In the condition of large batch size, CURAND can be several to tens fold faster than MKL for the same algorithms.

Modulo arithmetic based RNGs are less efficient than binary arithmetic ones, just as aforementioned. Combined Multiple Recursive algorithm (No. 7) and Wichmann-Hill algorithm (No. 9) present the slowest RNGs, followed by Multiplicative Congruential (No. 8). As a comparison, Mersenne Twister algorithm presents the fastest RNGs. CPU-based SFMT19937 (No. 12) can be one order of magnitude faster than CPU-based CMR32K3A (No. 7) while the GPU version (No. 4) can be 5 fold faster than the CPU implementation. Considering the good statistical property of MT [13], [14], [17], it makes the best RNG of all the RNGs concerned.

B. Solution Quality

In all experiments, 150 independent trials were performed for per function on each dimension, where 10,000 iterations were executed for each trial. 150 integer numbers were randomly generated from uniform distribution as seeds for each trail, and all RNGs shared the same 150 seeds. All particles were initialized randomly within the whole feasible search space and the initialization was shared by all RNGs (to be exactly, RNG No. 10 was used for the purpose of initialization).

The results (average values and standard deviations) are listed by Tab. III.

For a particular function, the solution quality can compared between any two RNGs with statistical test. But there is no direct way to compare a groups of RNGs (13 in our experiments).

1) League Scoring Strategy: To compare the results in a systematic and quantitative way, a league scoring strategy was adopted here. The results for two different RNGs, say A and B, on the same function of the same dimension are compared with $p = 0.05$ using rank-sum test. The scoring rules are illustrated by Tab. V. If A is better than B (i.e. $A < B$ assuming minimum problems), then A scores a points while B scores b points. On the contrary, if B is better than A, then B scored a points while A scores b points. Otherwise, it's a

TABLE II: RNG Efficiency Comparison Under Different Batch Size
(# of random numbers per nanosecond)

	GPU				CPU								
	CURAND				MKL								C
Batch Size	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0.3	0.3	0.3	0.3	17.5	17.3	14.8	15.9	7.8	15.2	14.7	15.2	
10	2.7	2.6	1.9	2.6	95.2	116.3	54.6	109.9	56.8	93.5	87.7	129.9	
20	4.8	4.9	3.2	5.1	144.9	166.7	67.1	192.3	79.4	181.8	161.3	227.3	
50	10.9	11.0	7.2	12.8	294.1	227.3	112.4	285.7	120.5	344.8	294.1	454.5	
100	21.1	21.6	14.2	25.8	400.0	263.2	138.9	357.1	142.9	434.8	384.6	714.3	
200	41.7	43.7	28.9	49.8	555.6	285.7	156.3	416.7	158.7	500.0	555.6	1111.1	
500	104.2	114.9	76.9	125.0	625.0	416.7	178.6	454.5	166.7	625.0	666.7	1428.6	
1000	200.0	243.9	163.9	232.6	666.7	555.6	185.2	454.5	172.4	666.7	769.2	1111.1	
2000	312.5	476.2	344.8	434.8	666.7	666.7	188.7	454.5	172.4	714.3	769.2	1250.0	
5000	500.0	1250.0	1000.0	909.1	769.2	769.2	192.3	476.2	172.4	833.3	833.3	1428.6	
10000	588.2	2500.0	1428.6	1250.0	714.3	833.3	192.3	476.2	175.4	833.3	833.3	1428.6	
20000	666.7	3333.3	1666.7	1666.7	714.3	833.3	192.3	476.2	175.4	769.2	769.2	1428.6	36.5
50000	769.2	10000.0	2500.0	3333.3	769.2	833.3	192.3	476.2	175.4	769.2	769.2	1428.6	
100000	714.3	10000.0	2500.0	5000.0	714.3	833.3	192.3	476.2	175.4	769.2	769.2	1428.6	
200000	714.3	10000.0	2500.0	10000.0	714.3	714.3	192.3	476.2	175.4	769.2	769.2	1666.7	

TABLE IV: Scores Achieved by RNGs for each Function

Function	D	GPU				CPU								
		CURAND				MKL								C
		1	2	3	4	5	6	7	8	9	10	11	12	13
Sphere	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	14	14	13	14	14	15	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	12	14	16	14	14	16	12	14	14
Elliptic	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	14	14	14	14	14	14	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	15	14	14	11	15	14	14	15	14	14
Discus	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	14	14	14	14	14	14	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	14	14	14	14	14
Ronsenbrock	10	14	14	15	14	13	10	14	15	14	14	15	13	17
	30	14	14	14	14	14	15	14	13	14	14	14	14	14
	50	15	14	14	14	12	14	14	14	14	14	14	14	15
	100	14	11	11	19	14	11	11	12	14	18	14	13	20
Ackley	10	14	14	12	15	14	14	14	15	14	14	14	14	14
	30	14	14	14	14	14	14	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	14	14	14	14	14
Weierstrass	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	13	14	14	14	13	14	17	14	14	13	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	14	14	14	14	14
Schaffers F7	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	14	14	14	14	14	14	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	14	14	14	14	14
Griewank	10	15	15	15	14	14	15	9	15	14	14	14	14	14
	30	14	14	14	14	14	14	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	14	14	14	14	14
Rastrigin	10	14	14	14	14	14	14	14	14	14	14	14	14	14
	30	14	14	14	15	14	13	14	14	14	14	14	14	14
	50	14	14	14	14	14	14	14	14	14	14	14	14	14
	100	14	14	14	14	14	14	14	14	15	13	14	14	14
total		505	502	500	512	498	499	498	505	505	508	504	502	514

tie so each scores c points. a, b and c satisfy the relation of $b < c < a$.

The scores calculated by the proposed method is presented by Tab. IV, where $a = 2, b = 0, c = 1$. For each row in the table, a maximum value can be picked out.

Observing Tab. IV and the corresponding Radar Map (Fig. 1), almost all cell were 14. Intuitively, it seems no significant disparity among all these RNGs. To analyze the performance in a more quantitative manner, the total scores were calculated (see the last row in Tab. IV). At most 16 points gap was observed among all the 13 RNGs. It is a very narrow gap considering that it's the total difference after $12*36 = 432$ rounds of 'competitions'.

TABLE V: Scoring Rules for Quality Comparison

Condition	Score of A	Score of B
$A < B$	$+a$	$+b$
$A > B$	$+b$	$+a$
Otherwise	$+c$	$+c$

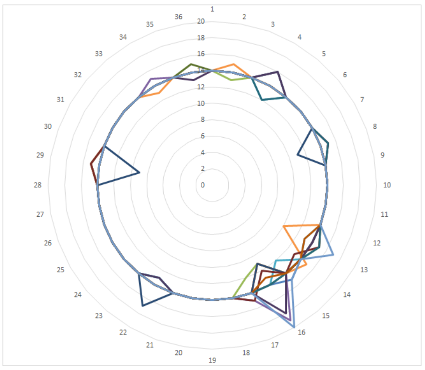

Fig. 1: Scores of All RNGs for Each Function

To make a detailed observation about if disparity exits for particular dimension or specific function, the scores were aggregated by dimension and by function respectively. Fig. 2 and Fig. 3 illustrate the aggregated results. No significant disparity was observed for these two conditions.

As a last comment on league score strategy, we shall take notice that the score-based comparison depend on the selection of scoring rules which determine to what degree a win be awarded and a lose be penalized. However, since what we encounter here is in effect a multi-objective comparison, there is no trivial optimal strategy without further knowledge. But the conclusion holds for common scoring rules, such as $a = 3, b = 0, c = 1$ and $a = 1, b = -1, c = 0$.

2) Lose-rank Strategy: To avoid determining the rational scoring rules, we proposed a new criterion named 'lose-rank' to compare the performance of multiple RNGs.

TABLE III: Results for Benchmark Functions

Function	D	1	2	3	4	5	6	7	8	9	10	11	12	13
Shpere	10	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)
	30	1.00E+02 (1.19E-03)	1.00E+02 (8.29E-04)	1.00E+02 (8.40E-04)	1.00E+02 (1.00E-03)	1.00E+02 (6.48E-04)	1.00E+02 (9.33E-04)	1.00E+02 (1.42E-03)	1.00E+02 (7.47E-04)	1.00E+02 (5.83E-04)	1.00E+02 (3.71E-04)	1.00E+02 (5.67E-04)	1.00E+02 (9.30E-04)	1.00E+02 (1.18E-03)
	50	1.01E+02 (2.06E-01)	1.01E+02 (2.82E-01)	1.01E+02 (2.71E-01)	1.01E+02 (2.56E-01)	1.01E+02 (2.59E-01)	1.01E+02 (2.22E-01)	1.02E+02 (2.51E-01)	1.02E+02 (2.52E-01)	1.01E+02 (2.45E-01)	1.01E+02 (2.40E-01)	1.01E+02 (3.28E-01)	1.01E+02 (2.38E-01)	1.01E+02 (2.89E-01)
	100	1.68E+02 (3.71E-03)	1.68E+02 (5.03E-03)	1.68E+02 (6.59E-03)	1.68E+02 (9.43E-03)	1.68E+02 (6.46E-03)	1.68E+02 (3.88E-03)	1.68E+02 (5.46E-03)	1.68E+02 (6.03E-03)	1.68E+02 (5.74E-03)	1.68E+02 (1.06E-02)	1.68E+02 (6.64E-03)	1.68E+02 (5.82E-03)	1.68E+02 (5.55E-03)
Elliptic	10	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)
	30	1.00E+02 (1.11E-05)	1.00E+02 (1.13E-05)	1.00E+02 (4.20E-05)	1.00E+02 (1.34E-05)	1.00E+02 (1.26E-05)	1.00E+02 (2.74E-05)	1.00E+02 (1.56E-05)	1.00E+02 (2.09E-05)	1.00E+02 (1.55E-05)	1.00E+02 (1.39E-05)	1.00E+02 (1.18E-05)	1.00E+02 (1.49E-05)	1.00E+02 (1.31E-05)
	50	1.09E+02 (1.57E+00)	(1.09E+02 (1.69E+00)	1.08E+02 (1.75E+00)	1.08E+02 (1.44E+00)	1.08E+02 (1.52E+00)	1.08E+02 (1.35E+00)	1.08E+02 (1.47E+00)	1.08E+02 (1.82E+00)	1.08E+02 (1.36E+00)	1.08E+02 (1.46E+00)	1.08E+02 (1.74E+00)	1.08E+02 (1.30E+00)	1.09E+02 (1.43E+00)
	100	4.46E+02 (1.02E-02)	4.46E+02 (1.07E-02)	4.46E+02 (1.02E-02)	4.46E+02 (1.14E-02)	4.46E+02 (8.95E-03)	4.46E+02 (2.39E-02)	4.46E+02 (7.79E-03)	4.46E+02 (1.38E-02)	4.46E+02 (1.06E-02)	4.46E+02 (9.73E-03)	4.46E+02 (9.27E-03)	4.46E+02 (5.59E-03)	4.46E+02 (6.89E-03)
Discus	10	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)	1.00E+02 (0.00E+00)
	30	1.00E+02 (3.88E-06)	1.00E+02 (6.91E-06)	1.00E+02 (5.93E-06)	1.00E+02 (5.84E-06)	1.00E+02 (7.92E-06)	1.00E+02 (5.37E-06)	1.00E+02 (3.93E-06)	1.00E+02 (5.96E-06)	1.00E+02 (6.52E-06)	1.00E+02 (6.62E-06)	1.00E+02 (7.35E-06)	1.00E+02 (6.24E-06)	1.00E+02 (6.01E-06)
	50	1.01E+02 (3.13E-01)	1.01E+02 (2.62E-01)	1.01E+02 (2.94E-01)	1.01E+02 (2.59E-01)	1.01E+02 (2.99E-01)	1.01E+02 (2.22E-01)	1.01E+02 (2.69E-01)	1.01E+02 (2.67E-01)	1.01E+02 (3.15E-01)	1.02E+02 (3.21E-01)	1.01E+02 (2.58E-01)	1.01E+02 (2.88E-01)	1.01E+02 (3.04E-01)
	100	1.68E+02 (1.13E-02)	1.68E+02 (5.76E-03)	1.68E+02 (1.01E-02)	1.68E+02 (6.42E-03)	1.68E+02 (5.09E-03)	1.68E+02 (7.81E-03)	1.68E+02 (5.41E-03)	1.68E+02 (7.07E-03)	1.68E+02 (6.15E-03)	1.68E+02 (8.30E-03)	1.68E+02 (1.02E-02)	1.68E+02 (4.76E-03)	1.68E+02 (6.64E-03)
Rosenbrock	10	1.05E+02 (1.27E+00)	1.05E+02 (1.79E+00)	1.05E+02 (1.54E+00)	1.05E+02 (1.60E+00)	1.05E+02 (1.47E+00)	1.06E+02 (1.20E+00)	1.05E+02 (1.88E+00)	1.05E+02 (1.55E+00)	1.05E+02 (1.84E+00)	1.05E+02 (1.65E+00)	1.05E+02 (1.63E+00)	1.05E+02 (1.29E+00)	1.05E+02 (1.88E+00)
	30	1.27E+02 (2.15E-01)	1.27E+02 (3.11E-01)	1.27E+02 (2.50E-01)	1.27E+02 (2.02E-01)	1.27E+02 (3.04E-01)	1.27E+02 (2.27E-01)	1.27E+02 (2.58E-01)	1.27E+02 (1.79E-01)	1.27E+02 (2.44E-01)	1.27E+02 (2.59E-01)	1.27E+02 (2.42E-01)	1.27E+02 (2.27E-01)	1.27E+02 (4.07E-01)
	50	7.54E+03 (2.00E+02)	7.61E+03 (3.03E+02)	7.58E+03 (2.03E+02)	7.57E+03 (1.86E+02)	7.65E+03 (1.86E+02)	7.60E+03 (2.07E+02)	7.57E+03 (2.16E+02)	7.61E+03 (2.22E+02)	7.54E+03 (1.97E+02)	7.58E+03 (2.47E+02)	7.59E+03 (2.59E+02)	7.61E+03 (1.99E+02)	7.57E+03 (2.25E+02)
	100	2.60E+04 (8.03E+02)	2.60E+04 (5.85E+02)	2.62E+04 (5.23E+02)	2.58E+04 (6.94E+02)	2.59E+04 (6.16E+02)	2.61E+04 (4.65E+02)	2.60E+04 (7.11E+02)	2.60E+04 (6.54E+02)	2.61E+04 (4.73E+02)	2.58E+04 (6.85E+02)	2.58E+04 (7.48E+02)	2.60E+04 (6.34E+02)	2.59E+04 (4.84E+02)
Ackley	10	1.00E+02 (2.16E-06)	1.00E+02 (3.31E-06)	1.00E+02 (3.48E-06)	1.00E+02 (3.88E-06)	1.00E+02 (2.96E-06)	1.00E+02 (6.75E-06)	1.00E+02 (3.07E-06)	1.00E+02 (4.17E-06)	1.00E+02 (3.05E-06)	1.00E+02 (6.34E-06)	1.00E+02 (2.74E-06)	1.00E+02 (2.82E-06)	1.00E+02 (4.27E-06)
	30	1.00E+02 (3.65E-02)	1.00E+02 (4.75E-02)	1.00E+02 (4.11E-02)	1.00E+02 (2.89E-02)	1.00E+02 (3.25E-02)	1.00E+02 (2.35E-02)	1.00E+02 (2.36E-02)	1.00E+02 (6.79E-02)	1.00E+02 (3.01E-02)	1.00E+02 (4.28E-02)	1.00E+02 (3.48E-02)	1.00E+02 (3.41E-02)	1.00E+02 (3.47E-02)
	50	1.16E+02 (9.11E-02)	1.16E+02 (8.76E-02)	1.16E+02 (8.13E-02)	1.16E+02 (9.58E-02)	1.16E+02 (6.69E-02)	1.16E+02 (9.67E-02)	1.16E+02 (7.95E-02)	1.16E+02 (1.16E-01)	1.16E+02 (8.20E-02)	1.16E+02 (8.14E-02)	1.16E+02 (7.89E-02)	1.16E+02 (9.40E-02)	1.16E+02 (9.84E-02)
	100	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)	1.17E+02 (5.15E+01)
Weierstrass	10	-2.00E+01 (3.70E-02)	-2.00E+01 (4.20E-02)	-2.00E+01 (9.90E-03)	-2.00E+01 (9.01E-02)	-2.00E+01 (1.24E-02)	-2.00E+01 (6.43E-02)	-2.00E+01 (5.78E-02)	-2.00E+01 (2.06E-02)	-2.00E+01 (5.77E-02)	-2.00E+01 (4.83E-02)	-2.00E+01 (1.08E-02)	-2.00E+01 (1.42E-01)	-2.00E+01 (6.41E-02)
	30	-5.82E+01 (4.43E+00)	-5.91E+01 (3.24E+00)	-5.93E+01 (2.87E+00)	-5.80E+01 (4.49E+00)	-5.94E+01 (2.12E+00)	-5.94E+01 (2.15E+00)	-5.98E+01 (1.48E+00)	-5.92E+01 (2.74E+00)	-5.77E+01 (4.95E+00)	-5.80E+01 (5.33E+00)	-5.90E+01 (3.53E+00)	-5.91E+01 (3.44E+00)	-5.84E+01 (3.89E+00)
	50	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)	-1.00E+02 (2.88E-14)
	100	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)	-2.00E+02 (2.88E-14)
Schaffer's F7	10	1.00E+02 (3.65E-03)	1.00E+02 (8.14E-03)	1.00E+02 (1.16E-02)	1.00E+02 (4.98E-03)	1.00E+02 (7.53E-03)	1.00E+02 (7.99E-03)	1.00E+02 (1.76E-02)	1.00E+02 (7.04E-03)	1.00E+02 (8.09E-03)	1.00E+02 (6.30E-03)	1.00E+02 (6.00E-03)	1.00E+02 (1.01E-02)	1.00E+02 (9.48E-03)
	30	1.01E+02 (1.79E-01)	1.01E+02 (1.98E-01)	1.01E+02 (1.38E-01)	1.01E+02 (1.73E-01)	1.01E+02 (2.07E-01)	1.01E+02 (1.59E-01)	1.01E+02 (1.83E-01)	1.01E+02 (1.65E-01)	1.01E+02 (1.45E-01)	1.01E+02 (1.46E-01)	1.01E+02 (1.85E-01)	1.01E+02 (1.79E-01)	1.01E+02 (1.88E-01)
	50	1.02E+02 (2.96E-01)	1.02E+02 (2.83E-01)	1.02E+02 (2.88E-01)	1.02E+02 (2.36E-01)	1.02E+02 (2.87E-01)	1.02E+02 (2.67E-01)	1.02E+02 (2.30E-01)	1.02E+02 (2.90E-01)	1.02E+02 (2.16E-01)	1.02E+02 (2.31E-01)	1.02E+02 (2.62E-01)	1.02E+02 (2.84E-01)	1.02E+02 (2.80E-01)
	100	1.44E+02 (2.09E-01)	1.44E+02 (1.92E-01)	1.44E+02 (2.37E-01)	1.44E+02 (2.10E-01)	1.44E+02 (1.68E-01)	1.44E+02 (1.82E-01)	1.44E+02 (2.43E-01)	1.44E+02 (2.03E-01)	1.44E+02 (1.78E-01)	1.44E+02 (2.52E-01)	1.44E+02 (1.76E-01)	1.44E+02 (2.55E-01)	1.44E+02 (2.03E-01)
Griewank	10	1.00E+02 (8.61E-03)	1.00E+02 (9.61E-03)	1.00E+02 (7.99E-03)	1.00E+02 (8.46E-03)	1.00E+02 (9.49E-03)	1.00E+02 (1.10E-02)	1.00E+02 (1.05E-02)	1.00E+02 (9.42E-03)	1.00E+02 (8.29E-03)	1.00E+02 (8.53E-03)	1.00E+02 (9.19E-03)	1.00E+02 (8.37E-03)	1.00E+02 (1.09E-02)
	30	1.00E+02 (4.29E-02)	1.00E+02 (5.48E-02)	1.00E+02 (5.47E-02)	1.00E+02 (4.25E-02)	1.00E+02 (4.89E-02)	1.00E+02 (4.80E-02)	1.00E+02 (5.32E-02)	1.00E+02 (5.69E-02)	1.00E+02 (4.88E-02)	1.00E+02 (4.71E-02)	1.00E+02 (5.07E-02)	1.00E+02 (5.04E-02)	1.00E+02 (5.66E-02)
	50	1.00E+02 (6.98E-02)	1.00E+02 (6.96E-02)	1.00E+02 (7.92E-02)	1.00E+02 (7.45E-02)	1.00E+02 (8.14E-02)	1.00E+02 (1.03E-01)	1.00E+02 (8.11E-02)	1.00E+02 (7.20E-02)	1.00E+02 (7.74E-02)	1.00E+02 (9.55E-02)	1.00E+02 (8.30E-02)	1.00E+02 (9.03E-02)	1.00E+02 (8.22E-02)
	100	1.03E+02 (1.49E-02)	1.03E+02 (1.24E-02)	1.03E+02 (1.39E-02)	1.03E+02 (1.33E-02)	1.03E+02 (1.38E-02)	1.03E+02 (1.68E-02)	1.03E+02 (1.70E-02)	1.03E+02 (1.62E-02)	1.03E+02 (1.37E-02)	1.03E+02 (1.41E-02)	1.03E+02 (1.12E-02)	1.03E+02 (1.50E-02)	1.03E+02 (1.85E-02)
Rastrigin	10	1.03E+02 (1.20E+00)	1.03E+02 (1.57E+00)	1.03E+02 (1.22E+00)	1.03E+02 (1.82E+00)	1.03E+02 (1.94E+00)	1.03E+02 (1.87E+00)	1.03E+02 (1.60E+00)	1.03E+02 (1.78E+00)	1.03E+02 (2.03E+00)	1.03E+02 (2.20E+00)	1.03E+02 (1.74E+00)	1.03E+02 (1.95E+00)	1.04E+02 (1.67E+00)
	30	1.30E+02 (5.51E-01)	1.29E+02 (8.14E-01)	1.30E+02 (4.64E-01)	1.29E+02 (7.81E-01)	1.30E+02 (6.14E-01)	1.30E+02 (9.05E-01)	1.30E+02 (5.53E-01)	1.29E+02 (8.75E-01)	1.29E+02 (7.41E-01)	1.30E+02 (5.28E-01)	1.30E+02 (4.96E-01)	1.30E+02 (7.24E-01)	1.30E+02 (6.15E-01)
	50	3.80E+02 (2.90E+01)	3.88E+02 (2.78E+01)	3.86E+02 (2.27E+01)	3.80E+02 (2.48E+01)	3.76E+02 (3.04E+01)	3.85E+02 (2.92E+01)	3.83E+02 (1.85E+01)	3.86E+02 (2.50E+01)	3.93E+02 (2.81E+01)	4.00E+02 (2.47E+01)	3.87E+02 (3.08E+01)	3.85E+02 (2.64E+01)	3.89E+02 (2.44E+01)
	100	9.12E+03 (6.59E+01)	9.14E+03 (5.06E+01)	9.13E+03 (5.99E+01)	9.13E+03 (5.44E+01)	9.12E+03 (5.90E+01)	9.14E+03 (5.85E+01)	9.12E+03 (6.05E+01)	9.11E+03 (6.10E+01)	9.12E+03 (6.10E+01)	9.13E+03 (7.26E+01)	9.13E+03 (5.97E+01)	9.12E+03 (5.62E+01)	9.12E+03 (5.94E+01)

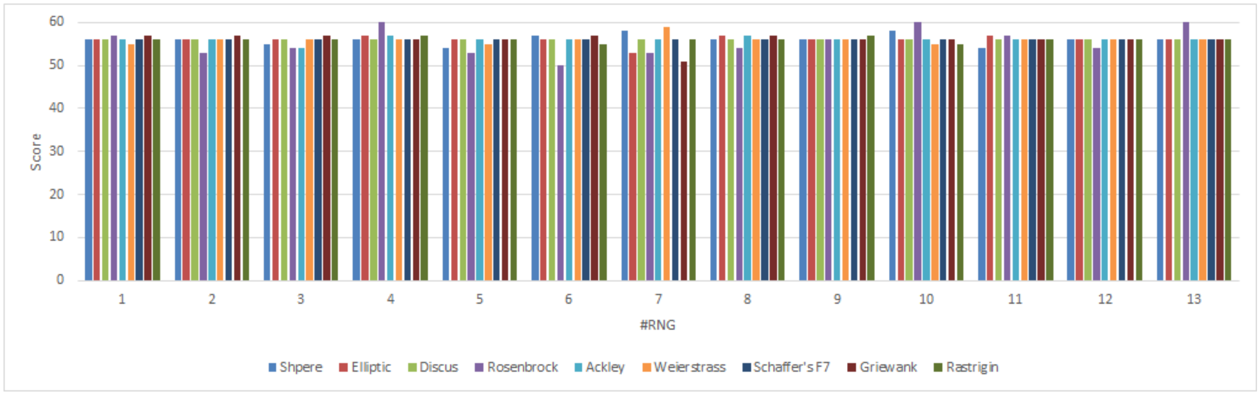

Fig. 3: Scores Aggregated by Function

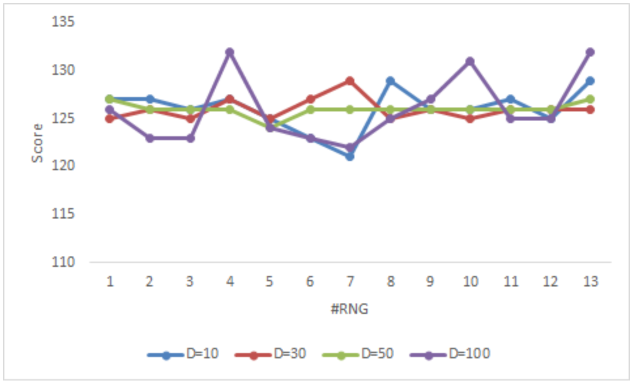

Fig. 2: Scores Aggregated by Dimension

Lose-rank can be calculated as follows. For A certain RNG, say R1, set its lose-rank to 0. R1 compares its solutions for a function with those of all other RNGs's one after another. If R1 statistically worse than some RNG, then add 1 to its lose-rank. In this way, we can calculate all RNGs' lose-ranks for all functions.

The idea underlying lose-rank is that if some RNG performs significantly worse in terms of solution quality, then it will has a relative large lose-rank.

The average lose-rank on all functions is listed by Tab. VI. The maximum lose-rank is about 1.6 (No. 4), which means any RNG, at its worst, is worse than less than 2 other RNGs. Considering each RNGs has 12 'rivals', it is a relatively minor lose-rank. The minimum lose-rank is around 0.4 (No. 5), which means that any RNG, at its best, will lose to some RNGs for certain functions. (Note No. 13 is among one of the worst in accordance with lose rank criterion while it achieved the highest score under league scoring criterion.)

The average lose-rank on each dimension and each function type is presented by Fig. 4 and Fig. 5, respectively. For a certain RNG, the lose-ranks can be lower or higher, but the fluctuation follows no remarkable pattern, and very high lose-ranks were observed rarely.

Based on all these observations, there exits no significant bad RNGs, and there is no outright good ones either. There is no strong reason to prefer any RNG to others as far as its impact on solution quality concerned.

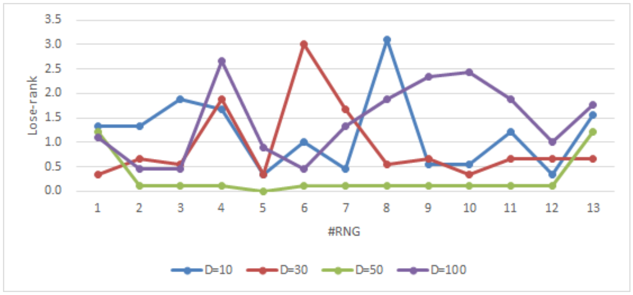

Fig. 4: Lose-rank Average on Dimension

V. Conclusions

Though different RNGs have various statistical strength, no significant disparity was observed in PSO in the experiments. Even the most common linear congruential generator performs very well, despite the fact that random number sequences generated by LCG are of lower quality in terms of randomness compared to other more complicated RNGs.

As a result, it is reasonable to utilize the most efficient algorithms for random number generation. In general, both CPU- and GPU-based RNGs can achieve best performance when generating blocks of random numbers that are as large as possible. Fewer calls to generate many random numbers is more efficient than many calls generating only a few random numbers. GPU-based RNGs can be several fold even one order of magnitude faster than their CPU-based counterparts, and Mersenne Twister algorithm presents the most efficient random number generator.

Only PSO using uniformly distributed RNGs was discussed in this work, however the two proposed strategies can be

TABLE VI: Average Lose-rank Values

	1	2	3	4	5	6	7	8	9	10	11	12	13
Lose-rank	1.0	0.6	0.8	1.6	0.4	1.1	0.9	1.4	0.9	0.9	1.0	0.5	1.3

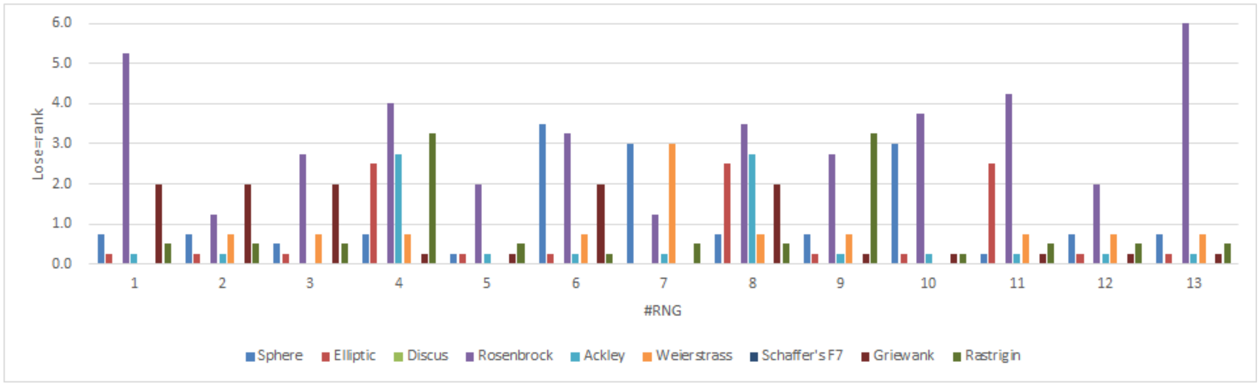

Fig. 5: Lose-rank Averaged on Function

extended to compare any stochastic optimization algorithm on any real-world optimization problems as well as benchmarks. RNGs for non-uniform distributions can also be researched in the proposed framework.

References

[1] D. E. Knuth, "The art of computer programming, volume 2: Seminumerical algorithms," *Amsterdam, London*, 1969.

[2] P. L'Ecuyer, "Random number generation," in *Handbook of Computational Statistics*, ser. Springer Handbooks of Computational Statistics, J. E. Gentle, W. K. Hardle, and Y. Mori, Eds. Springer Berlin Heidelberg, 2012, pp. 35–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21551-3_3

[3] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1268776.1268777

[4] M. Clerc, "List-based optimisers: Experiments and open questions," *International Journal of Swarm Intelligence Research (IJSIR)*, vol. 4, no. 4, pp. 23–38, 2013.

[5] C. J. A. Bastos-Filho, J. Andrade, M. Pita, and A. Ramos, "Impact of the quality of random numbers generators on the performance of particle swarm optimization," in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, 2009, pp. 4988–4993.

[6] C. J. A. Bastos-Filho, M. Oliveira, D. N. O. Nascimento, and A. D. Ramos, "Impact of the random number generator quality on particle swarm optimization algorithm running on graphic processor units," in *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*, 2010, pp. 85–90.

[7] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[8] L. Howes and D. Thomas, *GPU gems 3*. Addison-Wesley Professional, 2007, ch. Efficient random number generation and application using CUDA, pp. 805–830.

[9] J. A. Rice, *Mathematical Statistics and Data Analysis*. Belmont, CA USA: Thomson Higher Education, 2007.

[10] P. L'Ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Math. Comput.*, vol. 68, no. 225, pp. 249–260, Jan. 1999. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-99-00996-5

[11] The Numerical Algorithms Group Ltd, "Nag library manual, mark 23," http://www.nag.co.uk/numeric/fl/nagdoc_fl23/xhtml/FRONTMATTER/manconts.xml, 2011.

[12] P. L'Ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[13] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[14] M. Saito and M. Matsumoto, "Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, A. Keller, S. Heinrich, and H. Niederreiter, Eds. Springer Berlin Heidelberg, 2008, pp. 607–622. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74496-2_36

[15] ——, "Variants of mersenne twister suitable for graphic processors," *ACM Trans. Math. Softw.*, vol. 39, no. 2, pp. 12:1–12:20, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2427023.2427029

[16] S. Kirkpatrick and E. P. Stoll, "A very fast shift-register sequence random number generator," *Journal of Computational Physics*, vol. 40, no. 2, pp. 517 – 526, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999181902278

[17] M. Matsumoto and T. Nishimura, "Dynamic creation of pseudorandom number generators," in *Monte Carlo and Quasi-Monte Carlo Methods 1998*, H. Niederreiter and J. Spanier, Eds. Springer Berlin Heidelberg, 2000, pp. 56–69.

[18] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, april 2007, pp. 120 –127.

[19] Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 1493–1500.

[20] AMD Inc., "Core math library (acml)," http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

[21] "The boost random number library," http://www.boost.org/doc/libs/1_55_0/doc/html/boost_random.html.

[22] W. B. Langdon, "A fast high quality pseudo random number generator for nvidia cuda," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 2511–2514. [Online]. Available: http://doi.acm.org/10.1145/1570256.1570353

[23] Intel Corp., "The math kernel library," http://software.intel.com/en-us/intel-mkl.

[24] NVIDIA Corp., *CURAND Library Programming Guide v5.5*, July 2013.